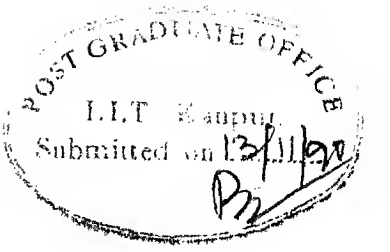
GOD : A GENERIC OBJECT ORIENTED DATABASE SYSTEM

A Thesis Submitted
in Partial Fulfilment of the Requirements
for the Degree of
MASTER OF TECHNOLOGY

By
V. VIDYAVATHI

to the
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY, KANPUR
NOVEMBER, 1990



CERTIFICATE

This is to certify that the Thesis work entitled GOD: A Generic Object_oriented Database system has been carried out by V VIDYAVATHI under my supervision, and has not been submitted elsewhere for the award of a degree.

T V Prabhakar

Station : Kanpur

Date : 13 th Nov, 90

(Dr. T.V. Prabhakar)
Assistant Professor,
Dept. of C.S.E.,
I.I.T. Kanpur.

Abstract

The object_oriented paradigm seemed to be an appropriate tool for the development of databases for CAD and other engineering applications, Software engineering environments and office information systems.

In this thesis we have implemented an object_oriented system. The model incorporates multiple inheritance, generic types, operators as properties of objects and operator overloading. A dynamic datastructure for the storage of objects has been proposed. Dynamic schema changes are allowed. Various semantical implications of the schema changes have been discussed. An object-oriented Language provides the interface to the system.

ACKNOLEDGEMENTS

Thanks is the cunning invention by humans to payback in a simplest way in return to the profits accumalated so far. If my inability to refrain myself from using this temptuous tool increases the toll of the victims to this majestic weapon it is not my fault. After all infinty plus one is infinity.

I thank Dr. T.V.Prabhakar for his guidance,encouragement and inspiration. His criticism and insight in pointing out the nuances of the problem always helped one in seeing the other sides of a multi faced structure called 'problem'.

I thank all my friends here and elsehere for adding color to life. I would also like to mention here my mates in the hostel for the unending discussions and their unabated vigor in carrying out debates which never ceased to defeat one's intention to remain a spectator.

I would like to thank C.S.E. staff and our dept. Librarian Ms. Sirin for their sinecere help in the closing chapters of my M.Tech. The memories of many others are documented well in the unerasable biological memory, nature's beautiful gift to mankind.

Table of contents

C h a p t e r 1

INTRODUCTION

The newly emerging applications like CAD, other engineering applications, office information systems and software engineering systems require databases whose underlying datamodel is more powerful than traditional datamodels. The applications of the old generation were less ambitious in their demands for the data types and the power of the query languages. The relational, network and hierarchial data models were powerful enough to match with these reqirements. The mathematical model for these models is complete. This made the design and implementation of the database an easier task. Compared to the traditional aplications, the present day applications vary a lot w.r.t.the degree of complexity in the semantics of the domains. As a result of this the gap between the application domain and the database programmer's domain, the impedence mismatch between the application development programming language and the database query language has also increased. This led to the demand for a new model fitting in to this complex environment.

When one tries to characterize these applications, the following important attributes emerge. Application development is very much a part of these databases. The direction is pointed towards the integration of code and structure. The degree of unification of behaviour with the state indicates the progress towards the utopian ideal. Reusing the code as an optimization

measure, incorporating dynamicity into the database to reflect the natural dynamic behaviour of the applicative domain, capturing the semantics of the applicative domain as far as possible, the flexibility for the user to model his objects with comparitively less translation burden--these constitute the basis of such a radical model. Some work has already been done in the semantic datamodelling field relating to these ideas.

The development of a datamodel is not independent of the query language interfacing the databases which are using that model. The query languages present a transparent view of the data base. Designing a query language capable of capturing the data model is of utmost importance. The query languages for these higher level data models are needed to be more powerful than the conventional query languages. They needed to incorporate powerful constructs like recursion. This motivated the integration of programming languages and database query languages leading to a new area of research, the Database Programming Languages. The expressiveness of these languages, the mathematical foundations which will help in studying the termination and correctness charactarestics and also their ability to capture the semantics of the data model are the important issues that are of concern to the researchers. Finally, the success of the model and the query language in decreasing the burden on the user in translating his real world view into the database domain decides the favourableness of such a model.

Different approaches have been taken in the process of

model reformulation. The first of these is to extend the relational model. The simplicity of the relational model compelled a group of people to retain the model and enhance its power through the extensions. Nested relational model[rk87] and Postgres[sr86] are the results of this. Adopting the reigning models in the programming language area to databases is another idea. Logic databases, Functional datamodel[gray] and object_oriented datamodel are the products of such a process. The other restricted data models being Geometric datamodels, Pictorial datamodels, datamodels for multimedia databases.

Of these the object_oriented model appeared to provide answers for many of the intriguing qustions. Abstraction, inheritance, dynamicity, added to the glamour of the model. In partcular it appeared to be a natural tool to realise a design environment. It's use can already be seen in CAD and kernel design [rj88]. But, the major problem that faced the researchers is the abstractness in the definition of the model itself. Fitting this model into a formal framework is an important and interesting task. The other important issue of concern is the practcality in the realisation of such a model. Many experimental prototypes and systems have been built and the performance issues and strategies in improving the performance are under study.

Related work: The arena of object_oriented databases can be divided into four branches. We will recount the related work according to this classification.

Model & its semantics: Type theories have been developed

to capture the semantics of the object oriented programming languages. A comprehensive survey in relation to the type theories has been done by [sc86]. They have talked about various type theories based on algebraic approach, record_based models and term_oriented models. They discussed about the semantical limitations of these models in capturing the subtyping relationships, including the polymorhic methods as part of the model. They also had talked about the bounded existential quantification as a tool for expressing the partially abstract objects.

A logical framework is used by [cg89] to reason about the objects. They translated the objects in to the first order predicates using tokens. Nested objects are represented with the help of tokens. Tuples will be replaced by the token ids and the values of these tuples themselves are represented by predicates. Sets are also represented in the same way. The nested terms are translated into unnested terms. They then used logical reasoning to do the required operations.

One more important issue thet needs attention is the study of the semantics of schema evolution in an object-oriented database system. In [jk87], a set of rules and invariants are stated to capture the important schema changes in an object_oriented database system. They used a simple foramal model called Property inheritance graph (PIG) to prove the soundness and completeness of the Orion taxonamy of schema evolution. In [jk187] the semantics of the model supporting composite objects

in an environment of dynamic schema changes and versioning has been stidied. They also proposed a locking strategy for the case of composite objects by introducing a set of new locks. The semantics of schema evolution has also been studied in [ps87]. They proposed some rules to preserve the invariants of the schema. The difference from Orion is that they allow single inheritance and automatic update of the schema in response to class modification. No dangling references are allowed. A nice overview of the research efforts in the direction of schema evolution and the propogation of changes is presented in [nr88].

Some of the other important ideas are as follows. Including the constraints as object types in the object-oriented paradigm will be a major step. If such a thing is incorporated the maintainance of view objects becomes effective. It increases the power of the language lot more. Another important extension in this direction is the automatic generation of programs given a set of constraints which will maintain the constrained object types.

Storage server: Objects are abstract, complex, independent, interactve, dynamic and can dynamically change their size. The storage of the objects is a complicated issue. Clustering the objects and caching the objects in the main memory is needed to improve the performance of the database. A strategy has to be evolved in clustering the objects on the secondary store. The aim is to map the conceptual proximity between the objects at the transaction level into a physical proximity while

storing them on the secondary store. So, the clustering needs information related to the semantical nature of the application. When shared objects are present in the database, replication of data may improve the performance. At the same time it increases the burden of the consistency maintainance of the system. There should be a proper compromise betwee the two. In some systems the user is given a handle to give advice to the storage manager regarding the clustering matters. Using the knowledge about the transactions in the storage scheme is of utmost importance. In [ck89], the knowledge about the structural and inheritanc relationships has been exploited in their clustering and bufferring schemes. In the encore database system [hz87], objects are clustered together in segments and segment groups. Replication of the objects and preloading objects are used to improve performance. Some relevant work has been done by [ws80] in their work on Relational Database schema design. They tried to enforce integrity through some automatically generated procedures instead of imposing it on the data maintainance if such an action led to the minimisation of the cost.

Another contribution to the improvement of performance comes from the usage of buffers. If one can predict the behaviour of objects in a transaction and if locality is observable then caching will be useful. One has to have an appropriate buffer replacement scheme to have an appreciable improvement. A context sensitive buffer replacement scheme has been used by [ck89]. They showed through the simulation experiments that this scheme with

prefetching performs best where as LRU with no prefetching performs the worst. When predictability is difiicult then pinning the objects by the user can be seen as one solution. One should also notice that the database buffer management is interlinked with the operating system buffer management [k88]. All these have to be taken into consideration while designing an efficient buffering scheme.

A B-tree based storage scheme has been proposed to store the objects in Exodus [cd89]. The B-tree is indexed by the byte position. This storage server is independent of the applications which are going to use it. The interpretation of objects is to be done by the application. Optimization of storage when shared objects are present and the sharing of storage in the case of versioning has been done effectively. A set_oriented storage of objects is done in the implementation of DASDBS kernel [we89]. Their aim is to store large,complex objects on the secondary memory as contiguously as possible. They used the chained I/O method in performing the operations on the secondary store.

Query optimization: Indexing the objects is needed to improve the performance of the objects. Two ways of indexing are possible. Single class indexing or class hierarchy indexing. The choice that is available in the optimization methods is 2_folded namely forward traversal and backward traversal. Orion[] merges both the approaches to optimize the performance.

Query Languages: Many object_oriented Programming languages came into existence recently. We will talk about the

prominent ones out of the lot. E++ [rc87], an extension over C++ [s84] to incorporates persistence and user_defined datatypes. O++ [ag89] is one more Language which provides declarative set constructs and persistent object types. An object identifier based Language IQL has been presented in [ak89] along with the semantics of the Language.

This completes a brief overview of the Related work in Object_oriented Databases.

The aim of the present thesis is to implement an experimantal prototype of an object_oriented system with persistence. In this process we intend to study the various semantical aspects of the model. Also the implementation difficulties in capturing the semantics of the model and improving the performace of the system are of interest to us. Finally, We wannt to use this system as a test base to study the adoptibility of the object_orinted paradigm for different sorts of application environments.

The thesis organization is as follows. Chapter 2 presents the characterestics of an object_oriented model and gives a brief summary of the Query Language used by the system. Chapter 3 illustrates various constructs of the language with some examples. chapter 4 details the implementatin aspects.

C h a p t e r 2

OBJECT_ORIENTED MODEL

There is no consensus among the database community about the object_oriented model. Though the philosophical aspects of the model are widely approved, the mathematical aspects of the model are not well formulated. In this chapter we will adopt a philosophical view in the process of our portrayal of the model.

2.1.OBJECT ORIENTED MODEL:

2.1.1.Object world: Essentially it is a homogeneous environment where everything is an object. All objects can be categorised in to two types. They are Type objects and Instance objects. A type object is a prototype of the instance objects. It contains information about the structural and behavioral properties of its instances. Type objects themselves are instances of a metatype 'OBJECT'. Every object inherits a property 'persistent' from the supertype OBJECT. This property will be used to distinguish persistent and non_persistent objects. The type objects along with the inheritance relationship acting on them form the schema of the database.

2.1.2.Object and properties: An object is an encapsulated data type. Structural and behavioral properties are attached to the objects. Objects are dynamic and independent. Therefore the properties can be added or deleted at any point of the time. In this way one can view the properities as descriptors rather than definitions. The structural properties are specified through

attributes and the behavioral properties are specified through methods. From the homogeniety aspect of the model it follows that both the attributes and methods are objects. In our model attributes and methods are system defined objects and they are not accessible to the user. This step is intended to provide a break to the circularity in the definition of the object.(see Figure 2.1)

```
define obj {
              [ int i;
                  [char c;
                   obj1 d;
                    [
                        .
                         .
                          .
                              [
                                obj2 q;
                              ] dn
                         .
                          .
                        .

                         ] d3{};
                       ] d2;
              ] d1;
           };
 define obj2 {

                   obj a1;
                  rest of the definition

              };
```

Fig. 2.1

An attribute has a name and a type. The attribute type can be any of the primitive datatypes or of the type of a type object defined earlier on the temporal scale. The primitive data types provided by our system are integer,real,charcter,string.

The Object definitions allow self recursion and mutual recursion (In Figure 2.1. you can see mutual recursion between obj and obj2). Besides the primitive data types two abstract data types set and tuple are present which can be used as constructors in composing a complex object. An object definition can be nested to any depth (See fig. 2.1.). An attribute can be selected by an **(root,path_to_attribute)** pair. root is the oid(oid is an abbrevation of object identifier whose discussion will follow shortly.) of the object from which the attribute has to be selected. **path_to_attribute** is a sequence of the attribute names on the path from the root to the attribute. All the attributes on the path_to_attribute sequence should be of the type either tuple or object.

A behavioral property (or method) has a name, body, the types of the parameters and the result type. The body of the method is to be defined in the same language as has been provided by the system. Parameters are to be supplied when the method accesses the attributes of objects other than self(self is the instance object which invokes the method). Method names should be unique within the object definition. The behavioral properties are selected by specifying **path_to_method** sequence. This sequence is similar to the path_to_attribute sequence except that the tail of the sequence is a method name followed by '(' parameters ')'. For example p.partno accesses the partno attribute of the object p and p.cost() accesses the cost method of the object p.

2.1.3.Object identity: Each object will have a unique

global identifier. These identifiers are called object identifiers (oid's in short) or surrogates. In the paper on the oid based language IQL[ak89], the semantics of the oid based operations has been studied. Usage of oid's helps in the sharing of structure, encoding of cyclicity, updates, and for making the query language more expressive. The problem of shared object representation will be simplified because redundancy can be avoided by storing the surrogates.

2.1.4. Inheritance: Inheritance is the most powerful feature this paradigm. When we say o_i inherits from o_j what we mean is that the object of type o_i will inherit all the propeties of object type o_j. Inheritance is transitive. Thereby the properties of an object type are in actuality a transitive closure of the properties formed at the object throgh inheritance. We wll illustrate this through an example.

```
define person {
                    string name;
                    [ string city;
                      int pincode;
                      string houseno;
                    ] address;
                };
define student : person {
                            int rollno;
                            course courses_taken_by {};
                         }
define teacher   : person  {
                             int salary;
                             course teaches {};
                           }
define ra : teacher student  {
                             }
```

Fig. 2.2

In the schema defined in Figure 2.2., any object of type ra will have all the attributes from the teacher and student object types. The object types student and teacher themselves inherit from a common object type person. So, ra will inherit the attributes from person also.

If a name conflict occurs between the properties inherited from two different super types, the properties of a super class relatively nearer to the object in the depth first order will take precedence over the other. For example consider the schema definition defined below.

```
define o { int i;                    define o1 : o {
        .                                    definition of the
        .                                    object
        .
        }                                    }

define o2 : {o  i;                   define o3 : o1, o2 {
            .                                    definition of
            .                                     the object
            .
          }                                    }
    o3 a;
```

Fig. 2.3.

Now if a.i is referenced then the definition of i in the object type o is taken as o comes first in the depth first order.

If the designer of the databse schema wants both the attributes to be inherited he should take care that a name conflict does not arise. In some systems the query language provides constructs using which the user can convey his priority to the system. In certain other systems aliasing is incorporated

so that an object can inherit both the attributes. But this increases the complexity of the schema manager.

2.1.5.Generic types: Generic types are synonymously called polymorhic types.

```
define element
        {
           abstract value;
         }
    abstract element.compare ();
        {
        }
    define collection
        {
           element dataset {};
        }
    define intelement:element
       {
            int value;
       }
    int intelement.compare (j)
     intelement j;
      {
         return (intelement < j);
      }
  define intcollection :collection
        {
           intelement dataset {};
        }
    collection.sort ()
        {
           element temp {};
           element i,j;
           temp = collection.dataset;
           if (i.compare(j) < Ø)
                    .
                    .
                    .
        }
    intcollection A;
    A.sort ();
```

Fig. 2.4

The schema of an object_oriented model displays inclusion polymorhism through the inheritance arc. Besides this a

parametric polymorhism is incorporated through the notion of Generic type. A generic type contains properties which are in reality abstractions of the properties of its subclasses (see Figure 2.4.). No distinction is made in the usage of these attributes from the concrete attributes(Conrete attributes are the attributes whose type will be known at the compile time.The subclasses of a Generic type which supply the abstract attributes defined in that Generic type are called oncrete subclasses.) The methods of the Generic type use these attributes in a uniform manner. One can notice that the methods which use the abstract attributes are polymorphic. The abstract attributes are bound to their types at run time.

One may want to use constraints to restrict the subclasses of a generic object type. For example, in the Collection example illustrated above one can restrict the subclasses of the object type element to only those classes which supply the attribute value and a method compare () taking two data elements as inputs. The specification and the implementation of such constraints is not clear.

2.1.6.Extendibility: The schema can be updated dynamically. The operations that can be performed are as follows.

(i) addition or deletion of an attribute

(ii) addition or deletion of a method

(iii) addition or deletion of an operator

(iv) addition or deletion of a type object

(v) addition or deletion of a type object from the super

types of a type object

Many consistency errors may result in the presence of dynamism. We will analyse the problem for the case of attribute deletion. For more details one can refer to [jk87]. The issues that are to be tackled when an attribute is deleted are as follows.

(i) Attribute deletion is transitive. So deleting an attribute should result in the deletion of the attribute in all the subclasses which have inherited the attribute. 'The attribute' is the key here. For example, in Figure 2.3 the deletion of the attribute i in o results in the deletion of the attribute from the object types o1 and o3. But the deletion of the attribute i in o2 results in the deletion from o2 only.

(ii) There can be methods which will be referencing the deleted attribute. In that case the methods have to deleted. The methods may be defined in the type object itself, or in its subclasses. If o_i is a subtype of a Generic type the attribute deletion may result in the disinheritance of that object from the Generic super type. One has to trade between the two invalidation methods namely immediate invalidation and lazy invalidation. The former approach requires a state to be stored relating the attributes and the methods which reference those attributes. The latter approach may result in postponing the task to the farther future if that method is not going to be accessed in the furthest future. Also one has to always screen the references at run time.

(iii) The changes have to be propagated to all the

instances.

In our system the changes are propagated to the schema objects and instance objects at the end of the transaction when all the persistent objects are stored back on to the secondary store. In the case of methods the only possible solution is to recompile them again at regular intervels. Presently we are not doing the recompilation. If an illegal reference is found with in a method definition at run time, that method will be deleted.

2.1.7.Versioning: The origin of this concept lies in the CAD and Software Engineering applications. For example, multiple copies of a design object may have to be maintained during the design process until a phase of the design is complete. The flexibility offerred encourages alternate thinking and gives the ability to store multiple copies of a design object until the final prototype of the design object is decided. Some of The issues raised in relation to the versioning paradigm are as follows.

(i) The addressability of the versions

(ii) Need to tag the objects as versionable at the time of object definition

(iii) What portion of the object will its version inherit?

(iv) What happens in the case of a complex valued attribute ?

Many more qustions have been raised in [ke89]. Versioning has not been implemented in the present thesis. With this we will

conclude the discussion of the model. In the following section we will present the salient features of our query language OQL.

2.2.QUERY LANGUAGE: Object Oriented Query Language(OQL) is an extention of C with object_oriented features and persistence. C has been chosen because of its popularity in the applicative programming environment and by extending an existing Query Language one can enjoy the benifits of the extended language as well as the power of the extensions besides the obvious release from the burden of the design of a complete language. In this section we will discuss the C_extentions.

Data types: The data types object,set,tuple are added.

The operations on an object data type are Instantiation, addition, updation, retrieval of the properties. The operations on a tuple data type are addition, updation, retrieval of the values of its attributes. The operations on a set type data type are

(i) The selection of the elements of the set satisfying a constraint

(ii) addition (deletion) of an element to (from) a set type object.

(iii) iteration over the elements of the set

As mentioned before the objects are broadly classified in to type objects and instance objects. Type objects are instances of a system defined object OBJECT. The user can not change the definition of the object OBJECT. The internal structure of the type objects is not accessible to the user. One can create the

type objects from the object definition interface provided by the query language. Similarly the addition of the values to the attributes of the type objects is done from an abstract declarative interface. The abstract declarative interface provided by the system facilitates the addition of attributes, methods, operators, super types to the schema objects.

Using this language one can create objects and manipulate them. Persistency is a part of the language. one can dynamically change the persistent characterestic of an object. Both the schema objects and instance objects are characterized by this quality. One more powerful feature of the language is that it allows ovrloading of the operators.

The features of the language are demonstrated through examples in the next chapter. The implementation details are discussed in the fourth chapter.

C h a p t e r 3

INTERFACE AND EXAMPLES

The user interacts with the system through an interface. The interface offers the following facilities.

1. Display all the object types in the system
2. Display an object definition in toto
3. Add object types to the database
4. Delete object types from the database
5. Add super classes to object type
6. Delete super classes from object type
7. Add attributes to object type
8. Delete attributes from object type
9. Add methods or operators to an object type
10. Delete methods or operators from an object type
11. Edit a method
12. Compile a method
13. Execute a method
14. Edit an OQL program
15. Compile an OQL program
16. Execute an OQL program

We have defined a query language OQL. OQL is an extension of C with object_oriented features. The object creation, schema changes can be done both from the interface level and the query language level. The methods, operators and the queries have to

be written in OQL. The constructs of the query language are specified below through a part database example.

Object definition:

```
Object part {

                int partno;
                string color;

            }

Object basepart: part {

                int mass();
                int cost();

                    }

Object composite_part: part {

                part components {};

                    }
```

Here ':' is a seperator between the object type and its super classes. In the above example basepart inherits from the object type part. That means it will inherit partno and color from its super types. An object type can have more than one object type as super.

Retrieving the properties of objects:

The properties of an object are referenced in a fashion similar to the access of fields in a record. If p is an object of type part, then its mass and cost are referenced by p.mass() and p.cost(). The partno of a part can be retrieved by p.partno. The behavioral property retrieval is distinguished from the structural property retrieval by suffixing it with '()'. if the method has parameters then they will be supplied in between the

paranthesis in the usual way of supplying the parameters.

Declaration of the variables: In the above example the variables of the type part can be declared as follows.

```
part partset{}; part special_part;
```

Attaching methods to objects:

```
basepart.mass ()
      {
          return (basepart.mass);
      }
basepart.cost()
      {
           return (basepart.cost);
      }
composite_part.cost()
      {
           int tempcost;
           for i in composite_part.components
               tempcost += i.cost();
           return (tempcost);
      }
```

similarly one can write the function for composite_part.mass().

In the definition of the composite_part.cost(), **composite_part** works as **self**. Whenever we refer to composite_part in the definition of any method attached to composite_part, it means that the instance which invokes that method is referred. When we say composite_part.components, the components of the instance which is invoking this method are retrieved.

Object **part** is a generic type. The object types base_part and composite_part are the concrete subclasses of this type. In the above mentioned example we have seen the way one can uniformly access the cost function irrespective of the type. This also demonstrates the natural incorporation of polymorhic functions in the object_oriented paradigm.

Addition of operators to an object: An operator should have a unique name within the object to which it is being added, the result type of the operator, the other operand on which this operator is going to operate,a priority value and an associativity value. The addition of operators to the objects is done in the following way.

```
part operator part <COPY,10,4> p
part p;
    {
      part.partno = p.partno;
    }
```

So when you say p copy q in your program, q.partno will be copied to p.partno.

The priorities of the operators are to be given relative to the priorities of the c operators. The priorities of the c operators have been fixed by the system. (these priorities are specified in the appendix) One can see where his operator will fall in between the already defined operators and depending on the position the user can fix the priority value.

Two different objects can have operators with the same name. In this way we will have overloaded operators. Operators are also inherited. They can be added and deleted dynamically.

Adding objects to a set type object:

Partset is a set type object. To this object the objects of type base_part as well as composite_part can be added in the manner mentioned below.

```
base_part temppart1;
composite_part p1, p2, p3, ..., pn,temppart2;
temppart1.cost = 200;
temppart1.mass = 100;
temppart1.partno = "p100a1";
      .
      .
      .
initialisation of p1, p2, ..., pn
      .
      .
      .
add p1 to temppart2;
add p2 to temppart2;
     .
     .
     .
add pn to temppart2;
add temppart1 to partset;
add temppart2 to partset;
          .
          .
          .
}
```

selecting parts from partset

```
{
    part heavyparts{};
heavyparts = get i in parts suchthat part.mass > 1000;
}
```

If one can use constraints as objects then one can define an object type heavy_parts which will contain only those parts which have a mass restriction on their mass. It will be interesting to see how one can automatically add objects to such a set given the constraint body (a set of constraints which

define a dependent object type). For instance, in the above mentioned example when ever a part is added to partset, it can be added to the set heavyparts depending on the value of its mass. If the mass of a heavy part is reduced then that part may have to be deleted from the heavy parts because now the mass value is less than the threshold value. If the attribute mass is deleted from the part object then heavyparts will loose their meaning. So the effects of all sorts of environment changes on a constrained object are worth studying. This is equivalent to the automatic updation problem of view objects.

Deleting objects from a set type object;

delete i in partset suchthat (part.supplier = "swami";

Iterating over a set type object: A declarative operator iterator is provided to iterate over all the elments of a set type object. We have already seen the usage of this construct when defining the method composite_part.cost(). Here we will give one more example.

```
count_of_heavy_parts(partset)
part partset{};
      {
        int count;
        count = Ø;
        for i in partset suchthat i.cost > 1Ø,ØØØ
             count+= 1;
        return(count);
     }
```

Changing the schema definition:

This will be done from the query language interface level. Let us say, in the part example one wanted to add to the part object the design of a part. He may add the following

attributes to the part object type.

```
string designfile;
part.display_design()
    {
```

body of the design program which uses the design filename and will display the design of the object. The design_file contains the information related to the object's design

```
    }
```

One can create a new object type 'exporters' as a subclass of the type suppliers which will exclusively contain the attributes, **countries** to which they will export and **items** they export and other related information.

Generic types in design: Suppose you are designing a metadatabase. The object **database** may contain a file_server, a query optimizer, a query language compiler etc. Now different instances of the object database may be having different file servers. This complicates the design of the object database. If generic types are not present, one has to define as many number of database objects as the number of file servers present in the domain. But in the presence of the Generic types one can accomplish this task by creating a Generic type fileserver and making all the concrete fileservers as subclasses of it. This generic type can be used in the definition of the object database. The generic fileserver will be providing an abstract interface to all the specific fileservers. The definition of the file servers and the the database object runs as follows.

```
define file_store
{
};
abstract file_store.addelement();
abstrct file_store.delete_element;
abstract file_store.search_element;

define db {

  filestore fs;
 ... other attributes ...

          }

object B_tree : filestore {

...      B-tree definition ...

}

object hash_table:file_store {

 ... hash_table definition ...

}
```

Initially, when one creates an instance database1 of the object type db, he has to specify the specific type of the fileserver of the instance database1. This specification should be done before the end of declarations part. This can be done as follows.

```
db database1; database1.fs is B_tree;
```

Persistence: An object can be declared persistent by prefixing it with the word 'persistent' in its declaration part. For example,

```
persistent part p;
```

All persistent objects will be stored on the secondary store. When the system is started all persistent objects will be loaded in to the main memory and when one exits from the system

all persistent objects will be stored back.

This completes the discussion of the language. In the next chapter we will talk about the implementation details.

Chapter 4

IMPLEMENTATION

We have implemented a main memory resident database system. The system has two constituent parts. One is the storage manager and the other is the translator. The storage manager maintains objects in the main memory and the secondary memory whereas the Translator maps the Oql constructs into functions on the storage objects. In the subsequent sections the services provided by these two components are discussed.

4.1.STORAGE MANAGER: The storage manager performs the following functions.

4.1.1.Object storage: Each object has a unique identifier called object identifier (oid). All objects are stored in an object table based on the oid of the objects. The storage scheme used is hashing.

4.1.2.Object storage format: The value representation of the object is in the following format.

```
|object|Ø|self        |1|1 st super's| ... |n|n th super's|
|size  | |attributes| | attributes |      | |attributes   |
```

Here the n th super's attributes contains the values of the inherited attributes from the n th super. The attributes are in the following format.

```
|attrno|attrvalue| ... |attrno|attrvalue|
```

Here the attribute no. is a system generated number which uniquely identifies an attribute within a type object. The

attrvalue is the value of the attribute which itself is in character string representation. The representation of the attrvalue is a function of the attribute type.

If the attribute type is of object type then the attribute value will be represented in the format mentioned above.

If the attribute type is of tuple type then the representation is as above except for the following difference: the fields corresponding to the super types are not present.

When the attribute is a set type attribute its value is represented as follows.

```
|set |element1|element2| ...   |element n|
|size|value   |value   |       | value   |
```

When the attribute is a primitive data type the values are mapped into the byte strings using a system defined mapping function. For example, an integer is represented as a 4_byte string. Every object is mapped on to a character string in the format mentioned above.

4.1.3.Object storage structure: Objects change their size dynamically. They can grow or they can shrink. This is because the objects are composed of variable size datatype 'set'. The other reason is that the attribute values of the objects can be null. To comply with this requirement of objects to have variable size records they are stored as a linked list of object chunks. An object chunk is a byte string and its size is predefined by the system. The size of a chunk remains constant throughout the life time of a database. Initially, when an object

is created, a chunk pointer is assigned to the object. A chunk pointer contains a pointer to a data chunk and pointers to the left and right chunks. Initially, when an object is created the data chunk pointer will be null. This step is intended to minimise the wastage of the space. When values are added to the object, data chunks will be allocated. Data chunks are allocated only when the object overflows from its object space. (From this point onwards we will use object space to refer to the physical storage object.) Similarly when values are deleted gaps will be formed in the chunks sometimes leaving complete chunks empty. Compression of the chunks is done to optimize the utilization of space. A data chunk will never have gaps in between the values. In other words every chunk will be partitioned into 2 parts. the first part contains the data and the second part contains empty space. That is why when a value is deleted the chunks containing that value are compressed. If the value occupies more than one chunk deletion of the value will create empty chunks. In such a case those chunks will be deallocated.

4.1.4.Schema object storage: Schema objects themselves are objects. They are the instances of a meta object OBJECT. The attributes inherited by them from OBJECT are object identifier, attributeset, superset,methset, operatorset and some other attributes needed by the system. All operations on objects are applicable to schema objects. They are also represented in the same format as any other objects.

The schema objects are stored in a separate table in the

main memory. This is done to decrease the contention problem between instance objects and schema objects while storing them in the hash table. In the secondary memory all instance objects are stored in an object file along with their object identifiers. similarly, all schema objects are stored in a separate file. No optimization or clustering is being done.

4.1.5.The object manipulation: The object manipulation functions are as follows.

Set/retrieve/modify the attribute values: The attribute whose value is to be set/modify/get and the instance in which the value has to be set is identified by an **<oid,attr_ref_string>** pair. When setting or modifying an attribute this pair along with the value which the attribute is going to take is passed to the storage manager where the required operation will be performed. Similarly when retrieving the values the <oid,attr_ref_string> pair is passed to the storage manager which in turn returns the value of the attribute. If the attribute type is a C data type permitted in the object definition, the character string representation of the attribute value is coerced back and returned. Otherwise if the attribute type is of object type an object pointer (the discussion of object pointer follows in the discussion of translator functions) is returned.

Now the task of adding (deleting) properties to (from) a schema object reduces to the addition of values to the schema objects.

Set manipulation: Elements can be added or deleted from a

set type object. A system defined library function set size is also provided.

Comparing two objects : No functions are provided to check the equality of two objects or two set type objects. One can easily write his own set equality routine using the iterator constructor. Similarly if one wants the deep equality between two objects he has to provide his own routine.

Assignment to an object: The assignment of an object value is translated to a object copy function. The type of the copy function is an object itself.

4.1.6.Persistency: Both schema objects and instance objects are characterised by the quality persistence. If an object is persistent it may impose persistency on some other objects. If an instance object is persistent then the schema object corresponding to the instance object should also be persistent. If the type of the attribute of a persistent object type is of object type then these object types have to be persistent. Similarly the methods defined on an object will have a result type. So if the object type is persistent the result types should also be persistent. This definition of persistence proceeds transitively. In the case of instance objects all the values of the instance objects are to be persistent.

A global persistent object set stores the oid's of all the persistent objects. Any object can be made persistent and any persistent object can be made non persistent during a transaction. Whenever an object is made persistent it's oid is

added to the persistent object set. Whenever a persistent object is made non persistent it's oid is deleted from the persistent object set. Objects are non persistent by default. If an object is not persistent by declaration and if it is in the persistency dependent list of a persistent object it will be made persistent by the system.

When the system is started all the objects are loaded in to the main memory assuming the availability sufficient space. when the transaction ends all persistent objects back on to the secondary store.

4.2. TRANSLATOR: The translator translates a program body in the query language OQL into the C code. A C data type **objpointer** is defined by the system. All objects will be of type objpointer at the C program level. All operations on objects are translated into functions on object pointers. We will discuss the functions of the translator in some detail.

Creating the schema objects: Whenever the translator comes across a type object definition it creates a schema object corresponding to that. This process consists of a number of steps. For each of the attribute definition with in the object definition, an attribute object is created and then added to the attribute set of the schema object. An attribute object contains the attribute name, attribute type, attribute number which is unique to the attribute with in the schema object and an attribute indicator which stores the information about the atribute being of set type or tuple type etc. If the attribute is

of object type then the attribute type will be the identifier of that object type. If the attribute is a tuple type attribute, then corresponding to the tuple definition an object is created and the object thus created is stored in the schema object table. Then this new tuple object's identifier is stored as the value of attribute type. This process continues recursively.

If a nonexistent type is referenced with in an object definition error will be reported and the attribute having this non existent type will not be added to the schema object. In the case of mutually recursive objects there is a possibility of mistaking a mutual recursion to be a reference to non_existent types. To avoid this mistake we proceed as follows. If the attribute of an object has a type which is not known at that point of time, we will assume the presence of mutual recursion. We will generate an identifier for this unknown type and a null chunk pointer is assigned to this and stored in the hash table. In this way we are creating null object types. The null chunk pointer indicates that the type object is in actuality a nonexistent object. Once a virtual object type is created, one can use this object type in the definition of other object types. But the instantiation of such a type is illegal. When that object is defined later, the null pointer will be replaced by the actual chunk pointer.

The null pointers have to be replaced before a function body starts. If such a thing is not done, it will be considered as a non_existent type. In such a case two lines of action are

possible. the first one is to delete the object types completely and the second one is to delete only the attributes having a non existent type. We are taking the second line of action. For this a temporary list of the names of the undefined objects is maintained. If the objects are defined later the name of that object type will be deleted from the list. Finally, when the data definition body ends, using this list the undefined object types will be deleted from the hash table. Later, when the attribute having this type is referred to in the program body, the attribute itself will be deleted as it is having an unknown type.

Translation of method definitions: The aim of this step is to translate the method definition in to an equivalent C function. The C function thus generated can not have the name of the method because it may generate ambiguity in the system. The reason for this is that two methods with the same name can be attached to two different objects. Therefore a unique function name is generated by the system and the translated method body will exist under this new name. This information and other information about the method like the result type of the method, method name are stored in a method object and the method object is then added to the method set of the object type.

The translation of operator definition also follows the same pattern. The operator object will contain the name of the operator, result type, priority and associativity values and the system generated function name for the operator. Then the operator object is added to the operator set of the object.

Similarly super classes are added to the super class set of the object.

Translation of variable declarations: Only the non_C variables are of concern to the system. When a declaration of such a variable is found, the variable will be added to a system maintained vartable. The vartable stores the type information as well as storage information related to the variables. Like C, the query language OQL is also a block structured language. Block variables will be added to the var table when the block begins and they will be deleted from the table when the block ends.

The index element of the iterator body is assumed to be a local variable of that body. So, no explicit declaration of the variable is needed. The type declaration of an index variable is implicit. Similarly, in the definition of methods and operators it will be assumed that a variable with the name of the object type on which the routines are being defined is existing. For example, if a method is being attached to the part object type, then a variable with the name of part is added to the var table. One can notice that this is similar to the self reference.

Translation of attribute references: At the oql level attributes are refernced through an object reference string. It consists of the name of the object followed by the attribute names and a '.' is used as separator. For eg. the part# of a part is accessed by part.part#. Now if the attribute part# is deleted then all the methods and operator function referencing this attribute will loose their meaning. If run time compilation is

done then some other problem arises. Let us say that after part# is deleted another attribute with the same name is added to the object. Now run time compilation will give a reference to this new attribute which is again not valid. to avoid this we are using a merge of these two approaches. We will compue statically the attr_ref_string corresponding to this attribute and we will compare this string with the dynamically computed attr_ref_string. If both of them match then that implies that the reference is legal.

Translation of method calls: Translation of method calls also folows the same pattern. A reference string is computed corresponding to the method reference statically and then it is compared with the method-reference_string computed at run time. If both of them match then the C function corresponding to this method is called.

Generic types: When translating the methods attched to a generic type, the static computation of attr_reference_string and meth_reference_string corresponding to the abstact methods and abstract attributes is not possible. Here the references will be bounded only at run time.

Expression handling: Object reference strings are used as terms in an expression. Object reference strings and method calls are used as terms. But the most important thing is operator translation. Initially an expression tree is formed using the priorities set by the c language. Now these priorities can be changed by the user. The same operator '+' can have two different

properties in two different object contexts. Some new operators like "EQUAL" may be added by the database designer to compute the equality of the objects. When the designer changes the priorities or sets priorities he has to give the priority value, the associativity value and the result type of the operation. The expression handler rebalances the expression tree using these values. Then the expression tree is traversed and translated into the equivalent set of storage level functions. The translation of the operators is similar to the method translation.

The translation of the other constructs like the iterator over a set object and add construct and delete construct is trivial.

Conclusions

We presented the important charecterestice of an object-oriented model and we proposed a query Language With object_oriented features to capture the model presented. A chunk based storage structure has been proposed for the storage of objects. The objects can be made persistent by the user through a programming language interface. Schema can be changed dynamically. The secondary storage scheme is a simple scheme. We assume that enough memory is availble on the main memory. All objects loaded in to main memory when the system starts and all the persistent objects are stored back on to the secondary store when the user exits from the system. In the process all the schema changes that took place during the system's life time are applied to the objects. Writing out all the objects on to a single file provides clustering.

Future work : The data model can be extended to incorporate versioning. Composite_inks can be added. The storage scheme itself can be modified in such a way such that oject types which are related through inheritance or through attributes reside on the same file otherwise they reside on a different file. The preprocessing step can detect the required files. Indexing the objects and clustering features can be added to the present system. This system can be used as a minimal system over which powerful database systems can be devaloped. A completely

different approach may be to concentrate on developing the mathematical model capturing the semantics of an object-orinted model.

References

[ag89] Agarwal, R. and Gehani, N.H. "ODE: The Language and the Model", SIGMOD 89

[ak89] Abiteboul, S. and Kanellakis, P.S. " Object identity as a Query Language Primitive", INRIA Report No. 1022.

[cd89] Carey, M.J., Dewitt, D.J., Richardson, J.E. and Shekita, E.G. "Storage Management for Objects in Exodus", Object_oriented Concepts, Databases and Applications, 341-370, Ed. by Kim,w. and Lochovsky, F.H., ACM Press, Newyork.

[cg89] Chen, Quiming. and Gardarin. G. " An Implementation model for Reasoning with Complex Objects", INRIA Report no. 793

[ck89] Chang, E.E. and katz. R.H. " Exploiting Inheritance and Structure Semantics for effective clustering and bufferring in OODBMS", 348-357, SIGMOD 89.

[gray] Gray, P.M.D. "Logic, Algebra and Databases", Affiliated East_west Press Pvt. Ltd., Newdelhi.

[hz87] Hornick, M.F. and Zdonik, S.B. "A Shared, Segmented Memory System for an Object_Oriented Database", ACM TOIS, 5, 70_95, 1987.

[jk87] Kim, w. and Jay Banerjee et. al. "Composite Object Support in an Object_oriented Database System", OOPSLA 87, Proceedings.

[jk187] Jay Banerjee and Won Kim. "Semantics and Implementation of Schema evolution in Object_oriented Databases", Sigmod 87.

[k88] Kim, K. " A Study of the Interactions between o.s. Memory Management and Database Buffer Management Strategies", Univ. of Illinois, Tr No. UIUCDCS_R_88_1446,88.

[ke89] Kent, W. "an overview of the Versioning problem", SIGMOD 89.

[nr88] Nguyer, G. and Riew, D. "Schema evolution in Object_oriented Database systems", INRIA Report No.

947,88.

[ps87] Penney, D.J. and Stein, J. "Class Modification in the Gemstone Object_oriented DBMS", OOPSLA 87, Proceedings.

[r88] Russo, v., Johnston and G. Campbell, R. "Process Management and Exception Handling in Multiprocessor Operating system using o_o design Technicques".

[rc87] Richardson, j.E. and Carey, M. "Programming Constructs for Database System Implementation", SIGMOD 87.

[rk87] Roth, M.A. and Korth, H.F. "The design of 1NF Relational Databases in to Nested Normal Form", SIGMOD87.

[s84] Stroustrup, B. "C++ Programming Language: Reference Manual", AT&T Technical Report No. 108, 84.

[sr86] Stonebraker, M. and Rowe, L.A. "The design of Postgres", 340-355, SIGMOD 86.

[ws80] Wons, H.K.T. and Shu, N.C. "An approach to Relational Database Schema Design", I.B.M. Technical Report No RJ2688 (34433), 1980.

[we89] Weikum, G. "Set_oriented access to large Complex Objects", SIGMOD 89.